Ethernut-GPIB Interface Manual

Raymond Meng Gao
Quantum Degenerate Gas Laboratory
Prof. Kirk W. Madison
Chem/Phys A015/023 UBC

August 24, 2005
Version 1.1.21
Contents

1 About this Manual ............................. 2

2 Introduction ................................. 3
   2.1 Architecture Overview .................. 4
   2.2 Introduction to Ethernut ................. 4
      2.2.1 How Does the Ethernut Work? ....... 6
      2.2.2 How Does Nut/OS Work? ............... 7
      2.2.3 How Does the FileSystem Work? ....... 8

3 Tutorial ..................................... 9
   3.1 Installing and Loading the Ethernut ....... 9
   3.2 Talking GPIB ................................ 12
      3.2.1 SSH Tunneling ......................... 12
      3.2.2 Connecting to an Instrument ........... 15
      3.2.3 Ethernut-GPIB Commands ............... 16
   3.3 Writing a Simple Program ................. 17
      3.3.1 Getting Java ........................... 18
      3.3.2 Sockets ................................. 18
      3.3.3 The Code ............................... 20
      3.3.4 Compiling and Running ................. 23

4 Hardware ................................... 25
   4.1 Design ...................................... 25
      4.1.1 NAT7210 ............................... 26
      4.1.2 Hooking up the CPU ................. 27
   4.2 Configuring the Board ..................... 29
      4.2.1 MMIO Address Jumpers ................. 29
      4.2.2 Other Jumpers .......................... 31
   4.3 Troubleshooting ............................ 31
      4.3.1 Ethernut Fuse .......................... 32
# Software

5.1 Software Architecture ........................................... 33
5.2 The Firmware ..................................................... 33
   5.2.1 Start of Execution ........................................... 34
   5.2.2 GPIB Server .................................................. 36
   5.2.3 GPIB Interface Functions .................................... 38
   5.2.4 NAT7210 and MMIO Programming ............................ 39
5.3 Building with Nut/OS ............................................. 40
   5.3.1 Using AVR C Makefiles ....................................... 40
5.4 Tips for Embedded Coding .......................................... 42
   5.4.1 Embedded Programs: Where does it end? .................. 42
   5.4.2 Using Flash ROM .............................................. 43
   5.4.3 Heap or Garbage Pile? ....................................... 43
5.5 Troubleshooting .................................................... 44
   5.5.1 Unresponsive DHCP client ................................... 44
   5.5.2 Floating Point Variables ..................................... 45

# To Do List

6.1 The Box ............................................................ 46
6.2 The Software ....................................................... 46
6.3 The Wiki ........................................................... 46

# List of References

A List of References .................................................. 47

# Ethernut-GPIB Commands

B Ethernut-GPIB Commands ........................................... 49

# Tutorial Code

C Tutorial Code ....................................................... 50
   C.1 EGInterface.java ............................................... 50
   C.2 Main.java ....................................................... 54
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Classic GPIB Architecture from the NI7210 Manual</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Ethernut-GPIB Architecture</td>
<td>6</td>
</tr>
<tr>
<td>3.1</td>
<td>SSH Tunnel</td>
<td>13</td>
</tr>
<tr>
<td>3.2</td>
<td>Setting up Putty for a GPIB-Server tunnel</td>
<td>14</td>
</tr>
<tr>
<td>3.3</td>
<td>Telnet to the GPIB-Server</td>
<td>17</td>
</tr>
<tr>
<td>4.1</td>
<td>Ethernut-GPIB Boards</td>
<td>25</td>
</tr>
<tr>
<td>4.2</td>
<td>The NAT7210 ASIC</td>
<td>26</td>
</tr>
<tr>
<td>4.3</td>
<td>Common GPIB Interface Design from the NAT7210 Manual</td>
<td>27</td>
</tr>
<tr>
<td>4.4</td>
<td>The Original Ethernut-GPIB TestBoard</td>
<td>28</td>
</tr>
<tr>
<td>4.5</td>
<td>Ethernut-GPIB design with GPIO Lines</td>
<td>29</td>
</tr>
<tr>
<td>4.6</td>
<td>Components on the Ethernut-GPIB</td>
<td>30</td>
</tr>
<tr>
<td>4.7</td>
<td>Filling the MMIO Address Jumpers</td>
<td>31</td>
</tr>
<tr>
<td>5.1</td>
<td>Ethernut-GPIB Software Architecture</td>
<td>34</td>
</tr>
<tr>
<td>5.2</td>
<td>Ethernut-GPIB Firmware</td>
<td>36</td>
</tr>
</tbody>
</table>
Chapter 1

About this Manual

Symbols used in this Manual:

- **Side Note - Extra information**

- **Caution - Important information that should not be overlooked**

- **Definition - Provides a quick definition of a term used**

Acknowledgements:

My best regards to everyone in the QDG lab during my workterm: Paul, Swati, Tao, Peter, Aviv, Janelle.

Special thanks to Dr Kirk Madison and Dr Bruce Klappauf who not only offered me this great job but helped me time and time again with all my projects.

More thanks to all people in the UBC Electronic Shop who helped build the Ethernut-GPIB: Richard, Tom, Gar, Marcel, and Stan.

And thanks to Kathy for drawing the Official Ethernut-GPIB icon. All the other note icons are Microsoft ClipArt.

Also thanks to the Egnite and the developers there for the helpful emails.
Chapter 2

Introduction

The Ethernut-GPIB Interface Project is as an alternative to purchasing expensive commercial GPIB controller boxes.

These similar commercial devices include Tektronix’s LAN-GPIB which costs about $1600 dollars CND and have compatibility limited to Windows

It is based on Egnite’s LAN enabled, microprocessor board, the Ethernut. It is an entirely Open-Source, Open-Hardware, Operating System Independent, Computer Independent solution.

Open-Source - All code is available for modification and redistribution including that of the Ethernut.

Open-Hardware - All hardware schematics and layouts are available for modification and redistribution including that of the Ethernut.

Operating System Independent - Writing software for this interface does not rely on a particular programming language or platform but only on the TCP/IP and Socket Protocol.

Computer Independent - Since the Ethernut runs on its own microprocessor and memory, no external computer is needed at all to keep it running.
2.1 Architecture Overview

The classic GPIB system usually consisted of a GPIB controller (e.g., PCI GPIB Card) and multiple instruments connected on the system. [Figure 2.1] It was limiting in several ways:

- The difference between a GPIB Controller and System Controller can be safely ignored in most cases. However, note that the Ethernut-GPIB was designed to be a monolithic System Controller; placing other Controllers/System Controllers in the same system may yield unpredictable results.

- Due to the cost and limited lengths of GPIB cables, the host computer must be placed physically close to the connected instruments.

- To allow data to be viewed from other computers, the host computer needs to run a data server of some sort.

The advantage of the Ethernut-GPIB [Figure 2.2] is its small size, portability, and separation from the computer. You can have several Ethernut-GPIB boxes each controlling a different group of devices running different experiments and oversee them on any computer from anywhere.

2.2 Introduction to Ethernut

This section skims over various basic aspects of the Ethernut that I felt was not clearly introduced in any of Egnite’s original documentation. For more in depth information about the Ethernut, refer to the series of manuals on the Ethernut’s homepage. (Also available in the List of References chapter in the Appendix).

The Ethernut is a series of Ethernet-Enabled Microcontroller boards built by the German company Egnite. With such features as Ethernet, Serial I/O, SRAM/flash ROM, GPIO lines (General Purpose I/O), and Nut/OS, the Ethernut can easily serve as a self-contained mini computer.
Figure 2.1: Classic GPIB Architecture from the NI7210 Manual
The entire project was developed with the Ethernut2.1. However, a new Ethernut3.0 is being developed by Egnite using the ARM7 CPU. Its compatibility with the current Ethernut-GPIB design is not known.

### 2.2.1 How Does the Ethernut Work?

It functions the same way as any computer:

1. Turn on Power
2. Begin Executing Instructions from Memory
3. Wait for User Input
4. Generate Output

The major difference is that the basic methods of input/output are not your traditional keyboard/mouse/monitor but limited to Serial ports, Ethernet and GPIO. The second notable difference is the types
of memory available on the Ethernut. Below is a brief description and comparison.

There are 3 types of memory on the Ethernut:

**Flash ROM** For storing program code and read only data; analogous to a computer’s hard-disk.

**SRAM** Runtime memory; analogous to any kind of RAM in a computer.

**EEPROM** For storing configuration data such as the MAC address; analogous to motherboard BIOS memory in a computer.

### 2.2.2 How Does Nut/OS Work?

The concept of operating system here is dramatically more primitive than any OS we can find on a modern computer. When we think of an operating system such as Windows or Linux, we might have a misleading image of an OS simply being a program running on our computers providing a basis of interaction between human and computer. However, what is often overlooked is the operating system’s responsibility of providing a programming interface to allow software control of the hardware. This is the essential duty of Nut/OS.

Counterintuitively, Nut/OS cannot ‘run’ by itself on the Ethernut. Nut/OS is a collection of libraries written in AVR C. You need an application that implements these libraries to see Nut/OS in action. The simplest application possible on Nut/OS looks like this in AVR C.

```c
#include <compiler.h>

int main(void) {
    for (;;) {
        for (;;);
    }
}
```

On the surface, this program runs in an endless loop doing nothing. However, since it included the directive, compiler.h, it is running on along with Nut/OS background procedures as event handlers and timers.

Furthermore, no real file system exists yet on Nut/OS. This means that there is no clear separation between operating system code (kernel) and application code, program instruction and data. You cannot
say, upload a new block of information onto the Ethernut’s memory and ask Nut/OS to remember where it is, know how to retrieve it, or modify it in a meaningful way. Thus, any structure about Nut/OS program, data or combination of these can only be modified externally. And modification can be only be done as a whole. Everytime we upload to the Ethernut, we are erasing its flash memory entirely and rewriting Nut/OS.

What is AVR C? AVR C is a modified version of C Language designed specifically for programming Atmel’s line of AVR microcontrollers.

2.2.3 How Does the FileSystem Work?

We might ask, if there is no file system, then how is Nut/OS applications capable of loading images and webpages etc? There is a mock file system called "UROM". An application called ”crurom" can take a file and generate an array in C code containing the hex representation of the file data. Thus, any required file data that we might need becomes static variables in the program. Ie: below is the beginning of the converted data form of the Ethernut logo image; it goes on for several pages.

```c
prog_char file1data[] = {
  0x47,0x49,0x46,0x38,0x39,0xa8,0x00,0x39,0x00,0xf7,
  0xff,0x00,0x00,0x00,0x00,0xc0,0xc0,0xc0,0xfe,0xfe,0xfe,
  0xfd,0xfd,0xfd,0xfc,0xfc,0xfc,0xfa,0xfa,0xfa,0xfa,0xfa,0xf9,0xf9,
  //...
};
```
Chapter 3

Tutorial

This is a quick start guide to using the Ethernut-GPIB including setting up the board, uploading the firmware and writing/compiling/running a basic Ethernut-GPIB application in Java™ on Windows and Linux. All the details here refer to Ethernut V2.1 and ENUT-GPIB firmware VR1.0. Since there is actually very little difference between the Windows and Linux tutorial, I will not make a separate section for each. My apologies to MAC users. I don’t know how to use one :-).

Windows specific details are in Red
Linux specific sections are in Blue

3.1 Installing and Loading the Ethernut

If your Ethernut is new and isn’t already loaded with the ENUT-GPIB firmware, this section will take you through the steps of uploading it manually. If this is your first time, please follow through the "Board Installation" tutorial in the Ethernut’s Hardware Manual first.

What is firmware? Firmware is a low level piece of software usually residing in flash memory of an embedded device. Here it refers to the software sitting on the Ethernut.

Before we can load our program onto the Ethernut, we need an interface to transfer data from our computer. The Atmega128 (Ethernut’s CPU) uses the JTAG interface (Joint Test Action Group) to load programs into its memory. The JTAG plug on the Ethernut is a 10
pin connector located beside the RS-485 screw terminal. The Ethernut Starter Kit provides a Serial-JTAG/ISP programmer that connects to the JTAG port on the Ethernut and a serial port on your computer. Due to the limited availability of serial ports on new computers, I used USB-Serial adapters instead. The problem with USB-Serial is that they come in so many brands and is difficult to find ones that have compatible drivers with Linux.

**Power on the Ethernut and Connect the Serial JTAG/ISP programmer to the Ethernut and the Computer; the green power LED should light up.**

Note it doesn’t matter whether the GPIB controller board is attached or not during this process. If you are using the Ethernut alone, it takes a 8-12V (~400mA minimum) supply. If you are using the Ethernut with the GPIB controller attached, the Ethernut will source power from the GPIB board. The GPIB board requires a 9V (~600mA minimum) power supply. **DO NOT attach power to both boards simultaneously.**

The programmer provided by Egnite’s Starter Kit uses the same connector for both JTAG and ISP (In System Programming) interfaces. As warned in the Ethernut’s Hardware Manual, **NEVER plug the ISP connector into Ethernut as it can damage the board.** The best precaution is to either tape it off or cut it off.

Next, we need a program to interface with the Serial-JTAG/ISP programmer. **Download the Nut/OS package from Ethernut’s homepage and extract it.** The tool we need is called “UISP” and its found under the nut/tools/win32 or nut/tools/linux directory. **Let’s add this directory to our system Path variable.**

Under Windows XP, we can do this by going to Control Panel → System → Advanced → Environment Variables and then appending the path (The Path of Ethernut Install)\nut\tools\win32 to the PATH variable with a semi-colon separator.

Under Linux, we can edit local path variables by adding the line
PATH=$PATH:(The Path of Ethernut Install)/nut/tools/linux to .bash_profile file. If the line already exists, simply append the path to the end with a colon separator. To add the path to all users, we must have root privileges and append the path to etc/profile.

Now goto the directory with our precompiled firmware hex file, GPIB.hex and enter the following command. Make sure the serial port is connected and assigned to COM1.

```
ui5p -dprog=stk500 -dserial=/dev/ttyS0 -dspeed=115200
-dpart=atmega128 --erase --upload if=GPIB.hex
```

What is a .hex file? A .hex file is the format that AVR’s C compiler produces, a pure binary.

If we get a message like this:

```
Error: No such file or directory
-> /dev/ttyS0
make: *** [burn] Error 1
```

Then either the Serial-JTAG controller is not connected properly, or the serial port driver is not installed properly, or it is not assigned to serial port COM1. If the latter is the case, we can use any other port we like by changing ttyS0 in the above command to ttySX where X-1 is COM port number.

If the procedure is successful, the Serial-JTAG LED will flash red/green as it begins to write to memory.

Next, connect the Ethernut to the network and connect the Ethernut’s serial port to the computer. (We can optionally disconnect the serial cable used for the JTAG programmer.)

Now we need a terminal emulation program to communicate with the Ethernut’s serial interface. For Windows, there is Hyperterminal or TeraTerm. For Linux, there is Minicom. Start the emulator with the following settings:

- baud: 38400
- no parity
- 8 data bits + 1 stop bit
- disable hardware flow control RTS/CTS
- disable software flow control XON/XOFF
The Ethernut manual describes that the Ethernut’s serial interface is compatible with any speeds within 38400 and 115200 but for some reason I’ve never gotten it to work with rates other than 38400 so all the baud rate variables written in my programs uses 38400 as well.

Hit the reset button on the Ethernut and we should see a message in the terminal like this:

```
Ethernut-GPIB 1.1.21 Server Started
Initializing Drivers...
Server Configured with Address: 172.16.1.239
<<INIT SEQ COMPLETE>>
```

Using the serial terminal is basic method of determining the IP address of the Ethernut.

### 3.2 Talking GPIB

This section describes all the ways of communicating with a GPIB instrument through the Ethernut-GPIB interface. However, this section does not explain the syntax of GPIB messages. Refer to GPIB tutorials or instrument manuals (Some are available in the Appendix of this manual). The GPIB.hex server program we just loaded in the previous section is running two sets of server threads: one set serving webpages on Port 80 and the other is the main method of communication with GPIB instruments; running a custom-protocol server on Port 15000. We will refer to this as the **GPIB-Server** throughout this document.

A "thread" in computer science is a task that can share resources and run simultaneously with other tasks within a program

#### 3.2.1 SSH Tunneling

If the Ethernut is connected on a local network (like on QDG) and we want to access it from a outside computer, the easiest way is to create a SSH Tunnel [Figure 3.1] (provided that the server computer is running an SSH server). This mini tutorial uses the example of connecting to an Ethernut’s GPIB-Server (address 172.16.1.239:15000) inside the network hosted by the server computer QDG.
Figure 3.1: SSH Tunnel

SSH Tunnel Diagram

Tunnel simulates a direct connection through localhost (127.0.0.1) Port XXXX

SSH Server running on qdg.physics.ubc.ca Port 22

Ethernet-GPIB server running on LAN address 172.16.1.239 Port:15000
To create a SSH tunnel on Windows [Figure 3.2], we can use any secure shell client that supports the tunneling option. Almost all do. I prefer PuttySSH, a popular, simple and free client for Windows. Open Putty and in the configuration dialog under connection→SSH, fill in the appropriate source port(any free local port) and the destination (the address and port of the Ethernut) and connect. Note that the ssh connection has to remain open for the tunnel to work.
To create a SSH tunnel on linux, simply use the -L option with the ssh command.

[-L port:host:hostname]

eg:

ssh qdg.physics.ubc.ca -L 12345:172.16.1.239:15000

This simply tells ssh to connect to the server qdg.physics.ubc.ca but at the same time forward our local port 12345 to the subnet address 172.16.1.239:15000. Note that the ssh connection has to remain open for the tunnel to work.

Now that we created the tunnel, the address localhost:(the tunnel port) becomes equivalent to 172.16.1.239:15000 if we were on a computer sitting inside the qdg network.

3.2.2 Connecting to an Instrument

The simplest way to see the Ethernut-GPIB in action is to open up our favourite web browser, hopefully Firefox ;-) , and type in the address of the Ethernut. Click on the GPIB Demo link and it will take us to a form that allows us to send commands to the GPIB instrument connected to GPIB Address 1. Connect a GPIB instrument, such as the Tektronix scope, to the Ethernut-GPIB and set it to Address 1. Try typing GPIB messages to it.

There is no internal mechanism for GPIB instruments to detect address conflicts. If two devices have the same address, and we try to query that address for example, they will both try to talk back simultaneously, resulting in a convoluted response.
What is GPIB Address? It is a unique number assigned to each GPIB instrument in the system. Usually, the controller will have address 0 and the rest is arbitrary. GPIB Addresses can only be changed physically on the instrument, not through software. Most systems are limited to 15 instrument but 31 addresses are available.

Simply type in any command and the program will try to obtain a response from the instrument. Of course, only query commands will generate a response.

Due to the limited memory available on the Ethernut, the servers are set to allow very limited number of connections simultaneously. For example, currently 4 is limit for the GPIB-Server.

Using the browser, we connected to the Ethernut-GPIB’s webserver on Port 80. Now we can try connecting to the GPIB-Server on Port 15000.

As an example, we can use telnet to do this. Telnet [Figure 3.3] is a classic TCP/IP command line protocol and program for sending raw text messages over the network. Telnet is almost exactly the same on Windows and Linux, with the basic format that looks like this.

\[
telnet \ [host] \ [port]\]

On the GPIB-Server, we can type messages to the GPIB instrument exactly the same way as in the browser. The only difference is that we can send Ethernut-GPIB commands.

### 3.2.3 Ethernut-GPIB Commands

Ethernut-GPIB Commands are a special type of message understood only by the Ethernut-GPIB. They allow the user to communicate with the either the Ethernut itself, or the GPIB controller board, as opposed to GPIB instruments. The rules are simple:

- Ethernut-GPIB commands begin with the ‘!’ character
- Ethernut-GPIB commands are always in capital letters
- All other messages are treated as GPIB messages and sent to the instrument directly.
Do not confuse ‘Ethernut-GPIB commands’ with ‘GPIB commands’. GPIB commands are a set of commands that all GPIB instruments must understand, as specified by IEEE488.2. They are also referred to as ‘Multiline Command Messages’.

For example, use the !SETIADDR command to change the reference GPIB address, allowing us to communicate with multiple instruments. [Figure 3.3] When we sent GPIB messages previously, we have been sending them to the default address, 1. However, to talk to a different address, we need to change the reference address. I.e:

!SETIADDR 19

Another command is !QUIT, this will simply disconnect us from the server. (We can also disconnect by sending an empty line) For a list of all Ethernut-GPIB commands, refer to the Appendix.

3.3 Writing a Simple Program

This section is a step by step tutorial on writing/compiling/running a simple program in Java that can communicate with a GPIB instrument through the Ethernut-GPIB interface.
3.3.1 Getting Java

All the resources mentioned here can be downloaded at java.sun.com. The sample code and sample program is available in the Appendix. If we goto the downloads section and select the latest version, (I used 1.5.0 for these examples) there will be several packages to select from. Here is what we need.

**JRE**  Java Runtime Environment, required to run java programs

**SDK**  Software Development Kit, required to compile and build java programs, already includes the JRE.

**Netbeans**  an IDE (Integrated Development Environment), this tutorial will not cover how to use Netbeans, but most of it is self-explanatory after installing.

So we only need to download and install the SDK package. On Windows, this is pretty straightforward. On Linux, the procedure is slightly different depending on the distribution. There are very helpful installation instructions on the downloads page, so they will not be covered here.

A quick check to see if at least the JRE is installed:

```
java -version
```

3.3.2 Sockets

This section gives some brief theory of Sockets. This knowledge is not required to be able to program with Sockets but nonetheless is useful information.

The concept of sockets is an universal idea in computer networking, not limited to Java. A socket simply consists of variables that contains the IP address and port of the server/client that we are connecting to. A socket is a type of header for a packet, which is a representation for groups of data being transferred over the network. As a real world example, consider a packet as a lettermail, and a socket as the part of the postage label containing the recipient’s address.

So what is IP address and Port? An IP (Internet Protocol) address is a number that identifies a computer on a network (a LAN, WAN,
or the entire Internet). The most common type of IP address we encounter is in the form of uuu.uuu.uuu.uuu in decimal or xx:xx:xx:xx in hexadecimal. This is referred to an IPv4 address. The IPv4 protocol allows for $2^{32}$ addresses. IPv6, an expanded address protocol that allows for $2^{128}$ addresses, is slowly replacing IPv4.

The IPv4 address is divided into several ranges, or classes. The following tables illustrates some important classes and subclasses.

<table>
<thead>
<tr>
<th>Class</th>
<th>Begin</th>
<th>End</th>
<th>Number of Networks</th>
<th>Number of Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10.0.0.0</td>
<td>127.255.255.255</td>
<td>128</td>
<td>16,777,214</td>
</tr>
<tr>
<td>B</td>
<td>128.0.0.0</td>
<td>191.255.255.255</td>
<td>16,384</td>
<td>65,534</td>
</tr>
<tr>
<td>C</td>
<td>192.0.0.0</td>
<td>223.255.255.255</td>
<td>2,097,152</td>
<td>256</td>
</tr>
</tbody>
</table>

The class divisions are assigned to accommodate different sized networks. As we can see from Table 3.1, the Number of Networks in an address range increases as the Number of Addresses increase. But the purpose of these kind of divisions is not apparent. However, out this concept developed 'netmasks', in which smaller ranges of address within these classes are used to create small private networks. [Table 3.2]

<table>
<thead>
<tr>
<th>SubClass of</th>
<th>Begin</th>
<th>End</th>
<th>Number of Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10.0.0.0</td>
<td>10.255.255.255</td>
<td>16,777,216</td>
</tr>
<tr>
<td>B</td>
<td>172.16.0.0</td>
<td>172.31.255.255</td>
<td>1,048,576</td>
</tr>
<tr>
<td>C</td>
<td>192.168.0.0</td>
<td>192.168.255.255</td>
<td>65,536</td>
</tr>
</tbody>
</table>

For example, by convention, seeing a computer with an IP address of 192.168.xxx.xxx will instantly tell us the following information:

- The computer is part of a small network of < 65536 members
-The computer does not have an exclusive IP address outside of this small network.

This allows each non-private IP address to host a network as large as 16,777,216 computers. However, each computer must be 'routed' through the non-private address to access the rest of the Internet.

On a smaller scale, this concept of multiplexing a single IP into many is exactly the idea of 'ports'. A port is a 16-bit integer representing the communication between a client to a host over a unique process or program. Without ports, a computer with one network card would only be able to run a single network program at a time. Many common ports will often identify its program. For example, a packet header containing port 80 will tell us that the protocol of transfer is http and a web-browser was most likely involved in sending this packet.

In Socket Programming, we will be dealing with the concepts of creating the socket (writing the address and making the package), connecting the socket (checking to see if the recipient exists), and writing/reading from the socket.

3.3.3 The Code

This section dives right into writing code in Java. If (OOP) Object-Oriented Programming or Java is new to you, that's okay, most of the code shown here will be very procedural and understandable. But it's never a bad idea to go through some basic Java tutorials on java.sun.com.

The purpose of this program is to connect to a Tektronix (3000 series) scope and grab some graph data from it.

Source files used in this example (available in the appendix):

- Main.java
- EGInterface.java

EGInterface.java is a class that encapsulates some basic functions for communicating with the Ethernut-GPIB through sockets. We can use this file as a starting point for most GPIB programs.
What is class and object? A class is the fundamental concept of Object-Oriented Programming. It is a template that lets the programmer design custom variables, or objects. Think of a class as a cookie-cutter and the object as the cookie. We mold the cookie-cutter into whatever shape once, then we can easily bake thousands of look-alike cookies. Yum.

Here [Table 3.3] is a brief description of the functions (in Java, functions are also called methods):

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EGInterface</td>
<td>String haddr,int port</td>
<td>Creates an EGInterface object</td>
</tr>
<tr>
<td>ibWrite</td>
<td>String s</td>
<td>Writes a message to the current GPIB address</td>
</tr>
<tr>
<td>ibReadC</td>
<td>StringBuffer s</td>
<td>Reads a message from the current GPIB address and store it in s</td>
</tr>
<tr>
<td>ibConnectGPIBAddress</td>
<td>int devaddr</td>
<td>Changes the current GPIB reference address to devaddr</td>
</tr>
<tr>
<td>ibFindInstrument</td>
<td>int devaddr,StringBuffer ID</td>
<td>Attempts find an instrument located at devaddr, if it exists, an identification string is stored in ID</td>
</tr>
<tr>
<td>ibCloseConnection</td>
<td>none</td>
<td>closes the socket and streams associated with this EGInterface object</td>
</tr>
<tr>
<td>getAddress</td>
<td>none</td>
<td>returns the current reference address</td>
</tr>
</tbody>
</table>

If we don’t already have the file, we can take the code from the appendix and copy it to a text editor and simply save it as EGInterface.java.

Now create a new text file called Main.java and start by adding these lines.
public class Main {

    public static void main(String[] args) {

    }
}

This is the starting point for execution in a Java program. Even though the Main class here serves no purpose, Java requires that ALL instructions in a program be encapsulated in a class, unlike C or other procedural languages.

Next we want to read the IP address and the port of Ethernut-GPIB from the command line. Add this code into main.

    if(args.length!=2) {
        System.err.println("Invalid Arguments");
        return;
    }

String IP = args[0];
int port = Integer.parseInt(args[1]);

Now we want to create a socket and connect to the Ethernut-GPIB on the given IP and port.

    EGInterface eg = new EGInterface(IP,port);

Next, let's connect to address 1. Make sure the scope is physically connected properly and its GPIB address set to 1.

    eg.ibConnectGPIBAddress(1);

To set the scope's address, we need to goto utility → System I/O → I/O → GPIB Talk/Listen → Talk/Listen Address and twiddle the knob beside the "select" and "coarse" buttons.

Notice that when a function is part of a class (such as EGInterface), we need to use the "dot notation" and say eg.myfunction(param 1,param 2); as opposed to simply myfunction(param 1,param 2).

Next, let's write a couple of GPIB commands that tells the scope to send us a waveform. These commands are specific to the Tektronix scopes and we can check the Tektronix Scope manual to see what they mean in detail.
eg.ibWrite("DATA:SOURCE CH1");
eg.ibWrite("DATA:ENCDG ASCII");
eg.ibWrite("DATA:START 1");
eg.ibWrite("DATA:STOP 500");
eg.ibWrite("CURVE?");

Note this will only grab 500 points from the current graph. If we want the entire graph, we need to either set "DATA:STOP" to 10000 points or decrease the resolution to 500 points.

Next, lets read the response and print it out.

    StringBuffer s = new StringBuffer();
eg.ibRead(s);
    System.out.println("The Response is: " + s);

and finally, close the connection.

    eg.ibCloseConnection();

Save this file as Main.java and make sure its in the same directory as EGInterface.java.

### 3.3.4 Compiling and Running

To compile, we will use the `javac` program. For the system to know where to find javac, we need to make sure the path jdk1.5.0.04\bin is in the system path. This is done in the same way as we added the Ethernut’s tools folder to the system path in the ”Installing and Loading” section of the tutorial.

Compiling is the same for all platforms. Goto the directory of your source file and type.

    javac Main.java

This will compile both Main.java and EGInterface.java as the Java compiler is smart enough to automatically recognize the dependency. After the compile is complete (assuming no error messages), javac will generate two new files: Main.class and EGInterface.class. .class files are to the Java Virtual Machine as executable binaries are to our computers.

Now to test our program giving it an IP and port.

    java Main [Ethernut’s IP] 15000
We should get something like this.

The Response is:
\[-34, -33, -32, -33, -33, -33, -32, -33, -32, -33, -32, -33, -32, -32, -33, -32, -32, -32\]

......
Chapter 4

Hardware

Figure 4.1: Ethernut-GPIB Boards

This chapter describes the design process of the Ethernut-GPIB and tips on building and debugging the board.

4.1 Design

The design of the Ethernut-GPIB Controller board is based around one piece of hardware: The NAT7210 GPIB Controller ASIC.

In theory, in order to build a GPIB controller we need to have thorough understanding of all the handshaking specifications. (See the
But since we live in an age of cheap reusable silicon technology, the NAT7210 ASIC (Application Specific Integrated Circuit) takes care of all the details for us.

The 7210 is actually an older model chip for GPIB controllers but fully satisfies our purposes. Newer options include the 9914 or TNT series.

4.1.1 NAT7210

This section summarize the functionalities of NAT7210 and focus on how it applies to the Ethernut-GPIB design. For an elaborate description of the chip itself, refer to the NAT7210 Manual (available in the Appendix).

The NAT7210 is a 40pin programmable IC designed to perform all the functions of IEEE 488.2 (GPIB) specifications. It is packaged with a 8 bit data bus and a 3-bit address bus and designed to be controlled by a CPU. The 7210 does not communicate directly with GPIB instruments but through 2 specific transceiver chips. [Figure 4.3] This is actually a common GPIB instrument design in the industry.

[Figure 4.3] shows the NAT7210 connected to the 75160 and 75162 transceiver chips which then connects to a standard GPIB cable. One might question the purpose of the chips upon seeing that most of the lines seems to go in from one side and directly out on the other. These transceiver chips serve two main purposes:

- Allows bidirectional data flow
- Buffers signal strength to meet IEEE488 specifications
The 75160 buffers the data lines and the 75162 buffers the handshaking lines. In the design of the Ethernut-GPIB, I used the 75161 chip instead, a simpler replacement for the 75162. The difference is the 161 assumes system controller state when instructed to become a controller while the 162 allows for an option to be either a controller or a system controller. This accounts for why the Ethernut-GPIB cannot co-exist with other controllers connected in the same system.

The system controller has the unique power to reset the state of all devices on the system and taking over control. There can only be one system controller in a system and my design will be implementing the system controller.

### 4.1.2 Hooking up the CPU

Since the NAT7210 is a programmable IC, we need a CPU to interface and program it. Enter Ethernut.

The Ethernut’s CPU, the Atmega128, has almost all of its pins routed to a large header on the side of the board. These include its data bus, address bus and GPIO lines. Thus the design of the control system can be done in two ways: using GPIO lines[Figure 4.5] or using
Memory-Mapped I/O with the Atmega128’s data/address bus. The first method I used in the testing stages [Figure 4.4] while the latter our electronic shop implemented in the finished product.

What is Memory-Mapped I/O? MMIO refers to the technique of the CPU directly addressing/writing to a peripheral device by treating it as an address in memory. This is demonstrated later in the Software section.

[Table 4.1] compares these methods.

<table>
<thead>
<tr>
<th>Pro</th>
<th>Con</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIO</td>
<td></td>
</tr>
<tr>
<td>- Easy to debug, full control of all lines</td>
<td>- Slower</td>
</tr>
<tr>
<td>- Requires no hardware logic</td>
<td>- More work to program</td>
</tr>
<tr>
<td>MMIO</td>
<td></td>
</tr>
<tr>
<td>- Requires logic analyzers to debug since bus is shared</td>
<td>- Faster</td>
</tr>
<tr>
<td>- Requires multiplexing between data/address buses</td>
<td>- Easy to program</td>
</tr>
</tbody>
</table>

Since MMIO is a cleaner solution and a standard concept of computer science, it was adopted in the end. The GPIO method was considered as a viable solution too because the Ethernut-GPIB was a small scale application and GPIB is a very slow protocol to begin with.
Note in the schematic, some GPIO lines from PortB of the Ethernut were connected to the INT,DACK,and DREQ lines. These functions were never used but could be implemented in the future.

4.2 Configuring the Board

This section provides tips on configuring a Ethernut-GPIB board’s hardware.

The Ethernut-GPIB is made of 3 physical boards [Figure 4.6]:

- The Ethernut
- The GPIB Controller Board
- The Bridge Board

The Bridge Board connects all the leads on the Ethernut’s header to the GPIB controller board. Note that the leads are also available on top of the bridge for testing and implementing new features in the future.

4.2.1 MMIO Address Jumpers

Since 8 bit of address space on the Ethernut needs to be mapped over to the NAT7210. We need an unused section of the Ethernut’s memory.
A chart of the Ethernut’s memory map (outdated) can be found on the Ethernut’s Memory Considerations Manual (Appendix) and can be used to single out a piece of memory. In the end, we choose the address

0xD0XX

Capital X represents don’t care bits since comparator in the schematic only cares about the most significant 8 bits. However, in reality only the addresses 0xD000 to 0xD007 are used.

How this is done on the GPIB Controller board is with 0 ohm resistors (jumpers). They are located at label I on [Figure 4.6]. A comparator compares the address coming from the Ethernut’s CPU and the address represented by a static logic pattern configured with these jumpers. If they match, that signifies the user is trying to address the GPIB Controller and the comparator will assert the CS pin on the NAT7210, allowing write/read to the NAT7210 (Actually, the cs pin merely allows the addresses on the NAT7210 to be modified).

As the jumpers short to ground (logic 0), we can carefully solder in a jumper wherever we want a 0. And since the leading 8 bits 0xD0 represents 11010000 in binary, we want the pattern to look like [Figure 4.7].
4.2.2 Other Jumpers

Besides the MMIO address jumpers, there are 2 other jumpers on the current version of the GPIB Controller board.

One is fairly obvious and that is the power LED lead at label H on [Figure 4.6].

The other is the Pullup-Enable jumper for the 75160 chip located at label J on [Figure 4.6]. This jumper decides whether the Pullup-Enable function (Refer to the 75160 data sheet) is controlled on board (shorted to ground) or by the NAT7210 (can be programmed). In most cases, this should be controlled by the NAT7210 but as PE is almost always pulled low, shorting the jumper gives the option of keeping it low.

4.3 Troubleshooting

The section describes known hardware problems with Ethermut-GPIB and should be kept open to additions. On way to resolve hardware
4.3.1 Ethernut Fuse

Symptoms:

The Ethernut blew up. The Ethernut died. The Ethernut is not powering.

Known Solutions:

There is a fuse on the Ethernut located at label B on [Figure \ref{components}] rated to blow at 1Amp but also protects against overvoltage by a selfdestructing mechanism when the voltage across shoots over 18 V. This problem concerns using the Ethernut without the GPIB Controller board. (The Ethernut fuse is bypassed since power is supplied from the GPIB Controller board).

The problem was that the Ethernut doesn’t come with a power supply (not even the starter kit) and only specified that it requires an input voltage between 8 and 12V. So we simply hoisted the nearest radio shack 12V DC adapter and plugged in. Long and behold it works. But occasionally it would blow the fuse because the standard power supply sends noisy voltage spikes at instant you plug it in. We measured the spikes on the scope and read levels up to 20 or 30 V, which surely would have blown the fuse. The solution was filter the power supply with one of the regulator circuits that Janelle Dongen built. It provided voltage regulation at a stable 12V and filters out RF noise with ferrite beads. The fuses are replaceable and manufacture by Littlefuse with catalog number 0453 001.
Chapter 5

Software

The chapter describes the design of the software architecture for the Ethernut-GPIB as well provide information on writing programs for the Ethernut in AVR C. These sections assume some previous knowledge of writing programs in C.

5.1 Software Architecture

The main aim of the Ethernut-GPIB software was to ensure portability. If someone decides that they like writing programs for one particular operating system, that is up to them. But it should be equally as simple for another person to write the same program for another OS.

The key to accomplishing this is using simple network protocols. For example, the GPIB server running on the Ethernut takes raw messages coming from the user and passes them onto the GPIB instrument verbatim. The concept is that all we need to know is how to use sockets and we can write programs in any language on any platform we want. Note in [Figure 5.1] there is a different “socket API” for every programming language but the concept is universal.

5.2 The Firmware

This section describes some of the important components in the Ethernut-GPIB’s firmware (software on board the Ethernut) and how they work.

Here is a list of the files and their purposes in source code of the firmware. [Table 5.1]

Since there are no classes in C language, here is the pretend hierarchy chart of the Ethernut-GPIB firmware. [Figure 5.2] The dependen-
cies of the files doesn’t look exactly like this but it outlines the idea I had in mind when designing. The next few subsections describe some of these modules in top down fashion.

### 5.2.1 Start of Execution

File: GPIBENUTKERNEL.c

Here the main function does several things:

- Initialize Ethernut Devices (LAN/Serial)
- Initialize the NAT7210
Table 5.1: List of Files in the Ethernut-GPIB Firmware Source

<table>
<thead>
<tr>
<th>File Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>7210INTF.c .h</td>
<td>Interface functions for the NAT7210</td>
</tr>
<tr>
<td>GPIB.c .h</td>
<td>Interface functions for GPIB</td>
</tr>
<tr>
<td>GPIBTCPS.c .h</td>
<td>Functions for the GPIB Server</td>
</tr>
<tr>
<td>DemoGPIBhttpd.c .h</td>
<td>Functions for the demo CGI and the http server</td>
</tr>
<tr>
<td>NutOSINTF.c .h</td>
<td>Initialization and Constants for dealing with NutOS</td>
</tr>
<tr>
<td>GPIBSetup.h</td>
<td>Stores several program wide constants</td>
</tr>
<tr>
<td>GPIBENUTKERNEL.c</td>
<td>Start of execution (main function)</td>
</tr>
</tbody>
</table>

- Start the HTTP Server Threads
- Start the GPIB Server Threads

In the code, multiple HTTP and GPIB Server threads are started.

```c
/*
 *create http service threads
 */
for (i = 1; i <= 4; i++) {
    char *thname = "httpd0";

    thname[5] = '0' + i;
    NutThreadCreate(thname, Service, (void *) (uptr_t) i, 640);
}

/*
 *create tcp service threads
 */
for (i = 1; i <= 3; i++) {
    char *thname = "tcpd0";

    thname[4] = '0' + i;
    NutThreadCreate(thname, ServeTCPQuery, (void *) (uptr_t) i, 640);
}
```

One might ask how the multithreading conflicts are resolved in this model. The answer is: what multithreading problems? Nut/OS implements **Cooperative Multithreading**, implying that a thread can only relinquish control **voluntarily**. Thus if we don’t need to
worry about resource sharing, locks, semaphores and other Preemptive Multithreading concepts.

What is Cooperative Multithreading? It is an old and simple approach to multithreading where threads can only voluntarily relinquish control. It is simple to implement but can run into problems very easily as the OS becomes more complicated. Now it is still used in many embedded applications such as the Ethernut.

5.2.2 GPIB Server

Files: GPIBTCPS.c GPIBTCPS.h

The GPIB Server uses Nut/OS TCP Socket API to create a server socket listening on port 15000 for connections and uses socket I/O
streams to read and write data. It is important to note that data is received via the `fgets()` function.

```c
fgets(buff, bufsize, instream);
```

`fgets()` is an old age C function that reads a single line of text from a file pointer. **Due to this fact, all clients MUST terminate their messages with a newline characters.** Otherwise, `fgets()` will block and wait for the end of the line. When program our clients, we must be aware that newlines are not the same on all operating systems. It is not sufficiently fool proof to use the C newline character ‘\n’. A standard symbol **CRLF** should be used instead. This a Carriage Return character followed immediately by a Linefeed character. When programming, the correct control sequence to use is:

```
"\x0D\x0A"
```

Note this is not always equivalent to:

```
\r\n
```

**Message Processing**

The GPIB Server sends all messages (anything that doesn’t begin with a ! character) to its message processor function. The message processor basically passes off the message to the GPIB device, tries reads a response, and sends it back. Since the Ethernut has limited memory, some GPIB responses (such as waveforms) are unreasonably large to store on the Ethernut even temporarily. Thus all the response data are streamed back to the client in discrete packets of 254 bytes. The server then terminates its responses with **CRLF**:

```c
fprintf_P(outstream, PSTR("\x0D\x0A"));
```

**Command Processing**

The GPIB Server’s command processor is designed to take inputs of forms similar to unix commands. Ie: **[Command] [-options] [arguments]** It is also written to be flexible for implementing new commands. To add a new command **XYZ**, follow these steps:
A - Add a new command ID  Add a new command identifier XYZ to the enumeration string in GPIBTCPS.h.

    enum {VERSION, SETIADDR, RESET, WRITE7210, READ7210, SENDCTRLCMD, XYZ};

B - Add the command string to comparator  There is a list of if statements in the beginning of the ProcessCommand function in GPIBTCPS.c. Their purpose is to match a string to a command. And a new else statement like this:

    ... 
    else if(strcmp(strupr(r),"XYZ")==0) {
        newCommand.commandName = XYZ;
    }
    ...  

C - Add a new command handler  There is a case statement that handles commands near the end of the ProcessCommand function in GPIBTCPS.c. Add a new case for the new command:

    case XYZ:
        //do something here
        return;
    break;

The commands options and parameters are stored in the newCommand structure. Its definition is in GPIBTCPS.h.

Note the command handlers could use separate functions, but since there aren’t many commands at this point, the case statement is simpler.

5.2.3 GPIB Interface Functions

Files: GPIB.c GPIB.h

The GPIB interface functions are all low level functions designed to communicate directly with the NAT7210. Thus most of these functions here involve register level programming, writing and reading integer values to registers, so understanding the code requires familiarity with the NAT7210 reference manual. However, below describes some important features
Init Sequence

The initialization sequence for the NAT7210 is in the `Init_Sequence` function. This function is called in the main function in `GPIBE-NUTKERNEL.c`. It contains calls to a set of macros defined in `GPIB.h`. These macros reflect the step by step initialization sequence documented in chapter 5 of the NAT7210 manual. Some of these steps in the sequence may need to be modified to accommodate changes in hardware. For example, two macros in combination define the system clock speed to be 20mhz, if the speed is ever changed, these macros need to be changed as well.

```c
#define SET_ICR Write_7210(0x2A,5) //20mhz
#define SET_ICR2 Write_7210(0x50,5);Write_7210(0x81,3)
```

Timeout Constant

There is a timeout constant in `GPIB.h`, currently set to 10000.

```c
#define TIMEOUT 10000
```

Many GPIB handshaking operations require some form of acknowledgement signal upon completion or failure. The NAT7210 does this by toggling certain bits in the registers. Thus in the code we need loops to constantly check for the status of these bits before and/or after certain operations. This timeout feature is implemented to keep the program from running into an infinite loop when an instrument is not responding for whatever reason. However, some operations take longer than others as GPIB is very slow, thus this timeout constant may need to be adjusted when implementing new GPIB interface functions.

5.2.4 NAT7210 and MMIO Programming

Files: 7210INTF.c 7210INTF.h

Being on the lowest point on the program hierarchy, the 7210 interface functions are actually the simplest. In face, it only comprises reading and writing using memory-mapped IO.

```c
//This is writing with MMIO
volatile unsigned int* p = (unsigned int*) (ADDR_OFFSET + addr);
*p = byte;

//This is reading with MMIO
volatile unsigned int* p = (unsigned int*) (ADDR_OFFSET + addr);
*byte = *p;
```
What is a volatile variable? The volatile keyword implies that the value of a variable will NOT be cached in the cpu registers for any reason. Caching in the registers allows for fast arithmetic operations, but this isn’t what we want for MMIO. If we are writing to a memory-mapped location, we want that the value be written directly to that location.

5.3 Building with Nut/OS

This section outlines how to install, build and compile programs under Nut/OS

The first order of business is to install the Nut/OS package. It is always best to download the latest version from the Ethernut’s Site. However, beware of the difference in versions. Many new changes are introduced in each new version and may become inconsistent with details in this manual. I used 3.9.5 but the latest available version as of now is 3.9.8. Luckily, all versions are available on Ethernut’s homepage and their CVS repository.

To be able to compile programs with Nut/OS, we need to first build the libraries. The Nut/OS configurator is a nice graphical interface for selecting various options for different builds. Instructions for installing and building the Nut/OS package are available in the Ethernut’s Software Manual.

5.3.1 Using AVR C Makefiles

The makefiles for AVR C look exactly the same as conventional makefiles for C. However, there are several common targets specific to building Ethernut and/or AVR C programs.

**Burn** is the name of a target that uses the **uisp** program to upload applications to the Ethernut.

**Webfile** is the name of a target that uses the **crurom** program to create the ‘urom’ source files from conventional webfiles such as html documents or images.
Program is the name of a target used by makefiles under the WinAVR environment to upload applications with the avrdude program. This is equivalent of uisp.

The easiest way to create a makefile for Nut/OS programs is to place our project folder under our build directory for Nut/OS and then simply include the Makedefs and Makerules that came with Nut/OS.

```make
include ../Makedefs
include ../Makerules

Now we can use some of the definitions such as $(LIBDIR) or $(SRCS:.c=.o).

This is bulk of the makefile for the Ethernut-GPIB firmware.

```make
PROJ = GPIB
TARGET = $(PROJ)
MCU = atmega128
WEBDIR = html
WEBFILE= urom.c

include ../Makedefs
include ../Makerules

SRCS = GPIBENUTKERNEL.c DemoGPIBhttpd.c GPIBTCPS.c $(PROJ).c\ 7210INTF.c NutOSINTF.c $(WEBFILE)
OBJS = $(SRCS:.c=.o)
LIBS = $(LIBDIR)/nutinit.o -lnutpro -lnutfs -lnutos -lnutdev\ -lnutos -lnutnet -lnutcrt
TARG = $(PROJ).hex
PARM = $(PROJ).eep

$(WEBFILE): $(WEBDIR)/index.html $(WEBDIR)/ASIC.jpg
$(CRUROM) -r -o$(WEBFILE) $(WEBDIR)

all: $(OBJS) $(TARG)
```
5.4 Tips for Embedded Coding

This section will describe some important differences between writing normal C code and embedded C code and guidelines for writing more efficient code. The reference manuals Nut/OS Memory Considerations and Nut/OS Threads, Events, and Timers have much more elaborate information. Please read these as well if you wish to optimize your programs.

5.4.1 Embedded Programs: Where does it end?

Unlike normal programs for computers, embedded programs can’t end. For example, consider the hello world program in C:

```c
int main(void) {
    printf("Hello World!");
    return 0;
}
```

On a computer, this program will simply print hello world and exit when it reaches `return 0`, which tells the operating system that the program is finished and safe to release its memory. On a microcontroller such as the Atmega128, the result of this program is catastrophe: might as well call it ‘Goodbye World’ instead. The problem here is that what happens after the CPU has reached the end of its instructions is unknown and is inconsistent from one microcontroller to another. The Atmega128 has a Watchdog Timer to prevent fatal system crashes due to unterminated code.

What is a Watchdog Timer? A Watchdog Timer is a common failsafe system placed on microcontrollers. It is simply a timer that will reset the entire system if it runs down to 0. Thus embedded compilers must always include instructions to ”pet the dog” (reset the timer) at appropriate instances to keep it from exploding.

The simple solution to avoid this is to ALWAYS put an infinite loop at the end of our programs.

```c
int main(void) {
    while(1) {
        // infinite loop
    }
}
```
printf("Hello World from the Atmega128!");
   while(1);
}

5.4.2 Using Flash ROM

Conserving memory is a large issue in embedded programming, much like normal programming when computers had 128k of RAM 15 years ago. The Ethernut provides 512k of Flash ROM (in addition to the Atmega128’s 128k Flash ROM) to allow larger programs. However this memory could also be used for storing static variables and long strings of text. (As opposed to storing them in SRAM at runtime).

AVR-GCC recognizes special a special keyword for this purpose: prog_char.
   For example, storing html code for CGI is waste of SRAM, but we can force it into Flash ROM like this:

static prog_char head[] =
"<HTML><HEAD><TITLE>Parameters</TITLE></HEAD><BODY><H1>Parameters</H1>";

   Additionally, avr libc provides an extra set of functions ending in _P as well as the macro PSTR() for putting static variables into flashrom. (Consult the avr libc manual for more details).

   For example:

fprintf_P(outstream, PSTR("x Invalid Parameters\n"));

   is much more conservative than:

fprintf(outstream, "x Invalid Parameters\n");

5.4.3 Heap or Garbage Pile?

The Ethernut’s software manual recommends extensive use of heap memory. I think this point is important enough to be commented on. (Please read the section called 'Heap Management' in the Ethernut Software Manual)
This refers to dynamically allocating memory using malloc and calloc
as opposed to declaring local variables. The Ethernut’s manual mentions wasting memory by declaring local variables in threads. However, since the stack size for a thread is pre-allocated to a fix size, we are not doing any harm by creating more local variables if the sum of the local variable sizes is well within this limit.

Additionally, there is a conservative and wasteful way of dynamically allocating memory in threads. Consider this snippet from the Ethernut-GPIB’s server thread:

```c
THREAD(ServeTCPQuery,arg) {
    //...
    NutTcpAccept(sock, TCP_PORT);
    //...
    long bufsize = 255;
    char* buff;
    if( (buff = calloc(bufsize,sizeof(char))) == NULL)
        printf("Out of Memory\n");
    //...
}
```

The server thread blocks at the `NutTcpAccept` function and waits for connections. After a user connects, it will allocate a piece of heap memory for its main stream buffer. Imagine if this allocation was done before the blocking call. Then the server will sit there with this huge chunk of memory already allocated doing nothing. Multiply the size of that buffer by the number of threads running and we have a wasted block of memory equivalent to declaring the buffer as a local variable.

The counter-argument is that allocating a heap at the beginning is faster. This is true but it is up to the programmer to decide whether speed or memory is more important. **The moral of the story is: heaps aren’t trivial either, think before you allocate them.**

## 5.5 Troubleshooting

This section will document known software bugs and their solutions. The best way to solve software problems with AVR C programming is read/post on the forums of **AVRFreaks.net**. There is a large community of AVR programmers (and freaks) there to help.

### 5.5.1 Unresponsive DHCP client

**Symptoms:**
Ethernut programs to refuse to obtain IP addresses assigned by the DHCP server. But when we checked the server logs, there appears to be corrupt packets coming from the Ethernut.

Known Solution:

Unfortunately, I still can’t conclude on this problem but it is most likely a hardware problem with the LAN IC possibly caused by ESD(electro static discharge). We ended up replacing the board.

5.5.2 Floating Point Variables

Symptoms:

Floating point related variables doesn’t work.

Known Solution:

Nut/OS does not need floating points and by default is not built to use them. You need to enable floating points in the configurator and rebuild Nut/OS. There is a tutorial on this on the Ethernut’s homepage under 'documents'.
Chapter 6

To Do List

This section describes Ethernut-GPIB related projects that are unfinished or not started. This chapter should be updated along as these projects progress.

6.1 The Box

The rack mount container for the Ethernut-GPIB is an unfinished project currently being worked on by the UBC Electronic Shop. I have no details at this point.

6.2 The Software

The GPIB Interface does not fully satisfy all the functions of a normal GPIB controller. Ie: it cannot perform parallel and serial polls. These functions should be implemented in GPIB.c and GPIB.h in the source file as needed. Also, the Ethernut-GPIB command set can be expanded to accommodate more functionality as needed.

6.3 The Wiki

This is a future project to start a wiki webpage for the Ethernut-GPIB depending on its popularity. The idea is to collect all the programs and libraries that people has written for the Ethernut-GPIB and to have them available online.
Appendix A

List of References

• Ethernut References
  
  **Ethernut Software Manual** Contains tutorials and extensive information on writing and compiling programs for the Ethernut.

  **Ethernut Hardware Manual** Hardware specifications; Schematics; Quick Start Guide; Configurations

  **Nut/OS Threads, Events and Timers** Information on Cooperative Multithreading structure of Nut/OS; Read this before using Threads; Declared Outdated by Egnite.

  **Nut/OS Memory Considerations** Information on memory structures on the Ethernut; Maps of memory spaces; Tips on conserving memory in code; Declared Outdated by Egnite.

  **Atmega128 Data Sheet** Comprehensive data sheets for the Atmega128 (Ethernut 2.x’s CPU); Contains sample code on writing/reading lines on the buses.

  **Nut/OS API Reference** References of Nut/OS libraries.

• GPIB References
  
  **Introduction to GPIB** A brief and concise introduction to the physical characteristics of the GPIB bus.

  **GPIB-ENUT Schematic** Working Revision of the GPIB-ENUT Controller Board.
NAT7210 Manual  Comprehensive reference for programming the NAT7210 Controller IC and building a GPIB Controller interface.

NAT7210 DataSheet  Brief data sheet for the NAT7210 chip.

NAT9914 Manual  Similar to the NAT7210 manual but for the 9914.


• WinAVR References
  
  Installing WinAVR  Quick guide to installing WinAVR, compiling, uploading, and using Programmer’s Notepad.

  AVRDUDE manual  Reference for WinAVR’s preferred programmer, AVRDUDE.

  AVR LIBC Reference  API reference to the AVR C library.

  AVR Freak  Lots of information on AVR programming; has a public forum.

• Java References
  
  Java Home  API references, tutorials, downloads; everything Java.
Appendix B

Ethernut-GPIB Commands

This is a list of the currently implemented Ethernut-GPIB commands

<table>
<thead>
<tr>
<th>Command Name</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETIADDR</td>
<td>!SETIADDR [addr]</td>
<td>changes current GPIB reference address</td>
</tr>
<tr>
<td>QUIT</td>
<td>!QUIT</td>
<td>disconnects from the GPIB server</td>
</tr>
<tr>
<td>VERSION</td>
<td>!VERSION</td>
<td>returns the version number of the Ethernut-GPIB</td>
</tr>
<tr>
<td>SENDCTRLCMD</td>
<td>!SENDCTRLCMD [cmd]</td>
<td>sends a GPIB Command through the system</td>
</tr>
<tr>
<td>WRITE7210</td>
<td>!WRITE7210 [addr] [cmd]</td>
<td>writes to a NAT7210 register, use with caution, make sure you’ve read the NAT7210 Manual first</td>
</tr>
<tr>
<td>WRITE7210</td>
<td>!READ7210 [addr]</td>
<td>reads from a NAT7210 register, use with caution, make sure you’ve read the NAT7210 Manual first</td>
</tr>
</tbody>
</table>
Appendix C

Tutorial Code

C.1 EGInterface.java

```java
import java.net.*;
import java.io.*;
import java.util.*;

public class EGInterface {

    private Socket sock;
    private BufferedReader in;
    private BufferedWriter out;
    private int devaddr;

    static final String SETIADDR = "!SETIADDR";
    static final String IDN = "*IDN?";
    static final long TIMEOUT = 2000;

    public EGInterface(String haddr, int port) {
        try {
            sock = new Socket(haddr, port);
            out = new BufferedWriter(new OutputStreamWriter(sock.getOutputStream()));
            in = new BufferedReader(new InputStreamReader(sock.getInputStream()));
        } catch (UnknownHostException e) {
            // Handle exception
        }
    }
```
System.err.println("Trying to connect to unknown host: " + e);
}
catch (IOException e) {
    System.err.println("IOException: " + e);
}

int getAddress() {
    return this.devaddr;
}

int ibWrite(String s) {
    if(!sock.isConnected()) {
        System.err.println("Socket is not connected");
        return -1;
    }

    try {
        out.write(s,0,s.length());
        out.newLine();
        out.flush();
    } catch (IOException e) {
        System.err.println("IOException: " + e);
    }

    return 0;
}

int ibRead(StringBuffer s) {
    if(!sock.isConnected()) {
        System.err.println("Socket is not connected");
        return -1;
    }

    try {
        
    } catch (IOException e) {
        System.err.println("IOException: " + e);
    }

    return 0;
}
Date d = new Date();

long time = d.getTime();
while (!in.ready()) {
   if((new Date().getTime() - time) > TIMEOUT) {
      System.err.println("Timed Out Waiting for Response");
      return -1;
   }
}
String t;
while( (t = in.readLine()).compareTo(""")==0 );

s = s.insert(0,t);
}
}
int ibConnectGPIBAddress(int devaddr) {
   try {
      Date d = new Date();
      String s = new String(SETIADDR + " " + devaddr);
      out.write(s,0,s.length());
      out.newLine();
      out.flush();
   }
   catch (IOException e) {
      System.err.println("IOException: "+e);
   }
   return 0;
}

String responseLine;
long time = d.getTime();
while (!in.ready()) {
    if (new Date().getTime() - time > TIMEOUT) {
        System.err.println("Timed Out Waiting for Response");
        return -1;
    }
}
responseLine = in.readLine();

if(responseLine.charAt(0) == 'x') {
    System.err.println("Error Trying to Change GPIB Address");
    return -1;
}

} catch (IOException e) {
    System.err.println("IOException: " + e);
}

this.devaddr = devaddr;
return 0;

int ibFindInstrument(int devaddr, StringBuffer ID) {
    int tempAddr = this.devaddr;

    if(this.ibConnectGPIBAddress(devaddr) != 0) {
        System.err.println("Unable to Connect");
        return -1;
    }
    try {
        Date d = new Date();

        out.write(IDN, 0, IDN.length());
        out.newLine();
        out.flush();

        long time = d.getTime();
        while (!in.ready()) {
            System.err.println("Timed Out Waiting for Response");
            return -1;
        }

        responseLine = in.readLine();

        if(responseLine.charAt(0) == 'x') {
            System.err.println("Error Trying to Change GPIB Address");
            return -1;
        }
    }
    catch (IOException e) {
        System.err.println("IOException: " + e);
    }

    this.devaddr = devaddr;
    return 0;
}
if( (new Date().getTime()-time) > TIMEOUT) {
    System.err.println("Timed Out Waiting for Response");
    return -1;
}
ID = ID.insert(0,in.readLine());
} catch (IOException e) {
    System.err.println("IOException: " + e);
}
if(this.ibConnectGPIBAddress(tempAddr)!=0) {
    System.err.println("Unable to Connect");
    return -1;
}
return 0;
}
void ibCloseConnection() {
    try {
        out.close();
        in.close();
        sock.close();
    } catch (UnknownHostException e) {
        System.err.println("Trying to connect to unknown host: " + e);
    } catch (IOException e) {
        System.err.println("IOException: " + e);
    }
}

C.2 Main.java

public class Main {
    public static void main(String[] args) {

if (args.length != 2) {
    System.err.println("Invalid Arguments");
    return;
}

String IP = args[0];
int port = Integer.parseInt(args[1]);

EGInterface eg = new EGInterface(IP, port);
eg.ibConnectGPIBAddress(1);

StringBuffer s = new StringBuffer();

eg.ibWrite("DATA:SOURCE CH1");
eg.ibWrite("DATA:ENCDG ASCII");
eg.ibWrite("DATA:START 1");
eg.ibWrite("DATA:STOP 500");
eg.ibWrite("CURVE?");

eg.ibRead(s);
System.out.println("The Response is: " + s);

eg.ibCloseConnection();
Bibliography


